Int J Softw Tools Technol Transfer (2016) 18:319–334
DOI 10.1007/s10009-014-0358-6

ICTSS 2013

Generating test case chains for reactive systems

Peter Schrammel · Tom Melham · Daniel Kroening

Published online: 15 November 2014

Abstract Testing of reactive systems is challenging because long input sequences are often needed to drive them into a state to test a desired feature. This is particularly problematic in *on-target testing*, where a system is tested in its real-life application environment and the amount of time required for resetting is high. This article presents an approach to discovering a *test case chain*—a single software execution that covers a group of test goals and minimizes overall test execution time. Our technique targets the scenario in which test goals for the requirements are given as safety properties. We give conditions for the existence and minimality of a single test case chain and minimize the number of test case chains if a single test case chain is infeasible. We report experimental results with our CHAINCOVER tool for C code generated from SIMULINK models and compare it to state-of-the-art test suite generators.

Keywords Test case generation · Reactive systems · Test optimization · Bounded model checking

1 Introduction

Safety-critical embedded software—for example in the automotive or avionics domains—is often implemented as a *reactive system* that periodically computes its new state and outputs as functions of the old state and some given inputs. These systems typically have to satisfy high safety standards, so tool support for systematic testing is highly desirable. The completeness of the testing process is often measured by defining a set of *test goals*, which are typically formulated as reachability properties. A good-quality test suite is a set of input sequences that drive the system into states that cover a large fraction of those goals.

Test suites generated by random test generators often contain a huge number of redundant test cases. Directed test-case generation often requires long input sequences to drive the system into a state where the desired feature can be tested. Furthermore, to execute the test suite, test cases must be manually chained into a sequence—or else the system must be reset after executing each test case. This is a serious problem in *on-target testing*, where a system is tested in its real-life application environment and resetting might be very time consuming [1].

This article presents an approach to discovering a *test case chain*—a single test case that covers a set of multiple test goals and minimizes overall test execution time. The essence of the problem is to find a shortest path through the system that covers all the test goals.

Example To illustrate the problem and our approach, we reuse the classical cruise controller example in [2]. There are five Boolean inputs: two for actuation of the *gas* and *brake* pedals, a toggle *button* to enable the cruise control, and two sensors indicating whether the car is *acc*- or *dec*elerating. There are three state variables: *speed*, *enable* (true when cruise control is enabled), and *mode*. The *mode* state records whether the cruise control is turned *OFF*, actually active (*ON*), or only temporarily inactive (*DIS*engaged) while the user pushes the gas or brake pedal.

This work was supported by the ARTEMIS VeTeSS Project and ERC Project 280053.

P. Schrammel (✉) · T. Melham · D. Kroening
Department of Computer Science, University of Oxford, Oxford, UK
e-mail: peter.schrammel@cs.ox.ac.uk

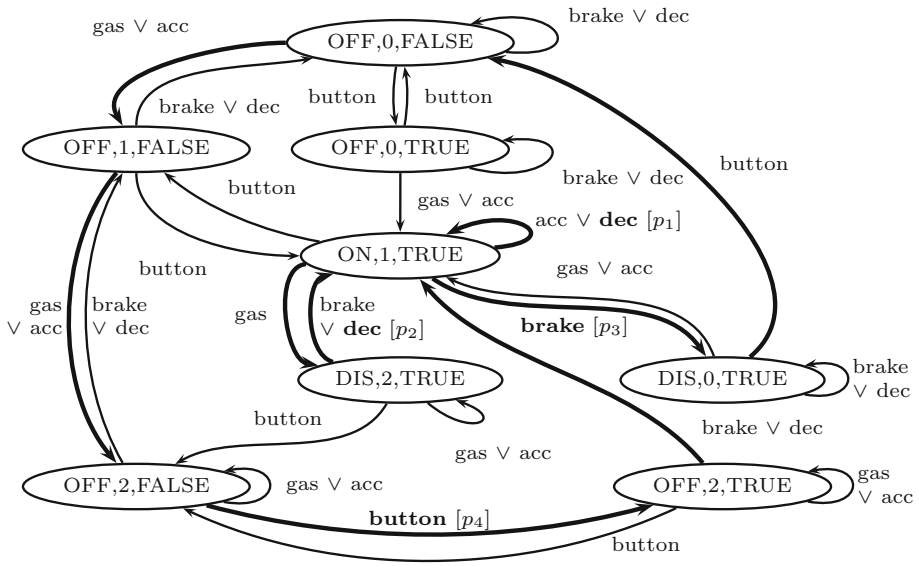
T. Melham
e-mail: tom.melham@cs.ox.ac.uk

D. Kroening
e-mail: daniel.kroening@cs.ox.ac.uk

Fig. 1 Code generated for cruise controller example

```
void init(t state *s) { s->mode = OFF; s->speed = 0; s->enable = FALSE; }
void compute(t input *i, t state *s) {
  mode = s->mode;
  switch(mode) {
    case ON: if(i->gas || i->brake) s->mode=DIS; break;
    case DIS:
      if( (s->speed==2 && (i->dec || i->brake)) || (s->speed==0 && (i->acc || i->gas)) )
        s->mode=ON;
      break;
    case OFF:
      if( s->speed==0 && s->enable && (i->gas || i->acc) ||
          s->speed==1 && i->button ||
          s->speed==2 && s->enable && (i->brake || i->dec) )
        s->mode=ON;
      break;
  }
  if(i->button) s->enable = !s->enable;
  if((i->gas || mode!=ON && i->acc) && s->speed<2) s->speed++;
  if((i->brake || mode!=ON && i->dec) && s->speed>0) s->speed--;
}
```

Fig. 2 State machine of the example. Edges are labelled by inputs and nodes by state $\langle mode, speed, enable\rangle$. Properties are indicated by the annotations $[p_1]$, $[p_2]$, $[p_3]$, $[p_4]$ and *boldface* input conditions. The *bold edges* show a minimal test case chain



A C implementation, with the sort of structure typical of code generated from SIMULINK models, is given in Fig. 1. Its state machine is depicted in Fig. 2. The idea is that the function compute will be executed periodically, for example on a timer interrupt. Thus, in this reactive program, there is a clear notion of a *transition* that relates to execution time.

We formulate some LTL properties for which we generate test cases:

$$p_1\colon\ \mathbf{G}\big(mode = ON \wedge speed = 1 \wedge dec \Rightarrow \mathbf{X}(speed = 1)\big)$$
$$p_2\colon\ \mathbf{G}\big(mode = DIS \wedge speed = 2 \wedge dec \Rightarrow \mathbf{X}(mode = ON)\big)$$
$$p_3\colon\ \mathbf{G}\big(mode = ON \wedge brake \Rightarrow \mathbf{X}(mode = DIS)\big)$$
$$p_4\colon\ \mathbf{G}\big(mode = OFF \wedge speed = 2 \wedge \neg enable \wedge button \Rightarrow \mathbf{X}\, enable\big)$$

In each of these properties, the operand of the **G** operator describes a specific transition in the state machine. These transitions are indicated in Fig. 2 by property-number annotations written next to the boldface labels on the four corresponding edges.

A *test case* is a sequence of inputs that determines a bounded execution path through the system. The *length* of a test case is the length of this sequence. A test case *covers a property* if it triggers the transition the property relates to. A *test suite* is a set of test cases that covers all the properties.

Ideally, we can obtain a single test case that covers all the required properties in a single execution of the program. We call a test case that covers a sequence of properties a *test case chain*. Our goal is to synthesize minimal test case chains—that is, to find test case chains with the fewest transitions. It is, however, not always possible to generate a single test case chain that covers all properties; multiple test case chains may be required. We will propose techniques for both situations.

We compute such a minimal test case chain from a set of start states *Init* via a set of given properties $\mathcal{P} = \{p_1, p_2, \ldots\}$ to a set of final states *Final*. For our example, with $Init = Final = \{mode = OFF \wedge speed = 0 \wedge \neg enable\}$ and $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$, we can obtain a test case chain that traverses the bold transition edges shown in Fig. 2. Beginning with *Init*, the state shown at the top of the diagram, this test chain first advances to cover p_4. It then covers p_1, p_2, and p_3

in sequence. Finally, it terminates in *Final*. One can assert that this path has the minimal length of nine transitions. Another minimal test case chain covers p_2 before p_1.

Testing problems similar to ours have been addressed by research on *minimal checking sequences* in conformance testing [1,3–6]. This work analyses automata-based specifications that encode system control and have transitions labelled with operations on data variables. The challenge here is to find short transition paths based on a given coverage criterion that are feasible, i.e. consistent with the data operations. Random test case generation can then be used to discover such a path. In contrast, our approach analyses the code generated from models or the implementation code itself, and it can handle partial specifications expressed as a collection of safety properties. A common example is acceptance testing in the automotive domain. Our solution uses bounded model checking to generate test cases guaranteed to exercise the desired functionality.

Contributions The contributions of this article can be summarized as follows:

- We present a new algorithm to compute minimal test chains that first constructs a weighted digraph abstraction using a reachability analysis, on which the minimization is performed as a second step. The final step is to compute the test input sequence. We give conditions for the existence and minimality of a single test case chain and propose algorithms to handle the general case.
- We have implemented a tool called ChainCover[1] for C code generated from Simulink models, on top of the Cbmc bounded model checker and the Lkh travelling salesman Problem solver (or alternatively, the Clingo Answer Set Programming solver).
- We present experimental results to demonstrate that our approach is viable on a set of benchmarks, mainly drawn from the automotive industry, and is more efficient than state-of-the-art test suite generators.

This article is an extended version of the ICTSS 2013 conference paper [7]. In addition to the core ideas published in [7], this extended article proposes an alternative to abstraction refinement using path constraints and an ASP solver (Sect. 4.2) and gives details of the generation of multiple chains (Sect. 4.3). We also report additional experimental results using the ASP solver and benchmarks requiring multiple chains (Sect. 6). Finally, we have provided several detailed proofs and the pseudo-code of the algorithms, which were omitted in the conference paper.

[1] http://www.cprover.org/chaincover/.

2 Preliminaries

Program model. A *program* is given by $(\Sigma, \Upsilon, T, Init)$ with finite sets of states Σ and inputs Υ, a transition relation $T \subseteq ((\Sigma \times \Upsilon) \to \Sigma)$, and a set of initial states $Init \subseteq \Sigma$. Without loss of generality, we can assume that $Init$ is the singleton set $\{S_0\}$; a system that chooses its initial state non-deterministically can be modelled with the help of additional inputs and an initialization transition.

We characterize (sets of) states and inputs symbolically by predicates. The transition relation is represented as a predicate $T(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s}')$ over vectors of state variables $\boldsymbol{s}, \boldsymbol{s}'$ and a vector of input variables $\boldsymbol{i}$. A valuation S (respectively S') of $\boldsymbol{s}$ ($\boldsymbol{s}'$) is called the prestate (poststate) of the transition. Valuations of $\boldsymbol{i}$ are denoted I. Similarly, the initial states are represented as a predicate $Init(\boldsymbol{s})$.

An *execution* of a program is a (possibly) infinite sequence of transitions $S_0 \xrightarrow{I_0} S_1 \xrightarrow{I_1} S_2 \to \cdots$ with $Init(S_0)$ and for all $k \geq 0$, $T(S_k, I_k, S_{k+1})$. Note that the execution semantics is deterministic in the sense that a sequence of inputs determines exactly one execution, which is an important requirement to ensure the repeatability of tests.

Properties. We consider partial specifications given as a set of safety properties $\mathcal{P} = \{p_1, \ldots, p_{|P|}\}$ that are written in a fragment of linear temporal logic (LTL) of the form $\mathbf{G}(\Phi)$, where Φ characterizes a set of finite paths. Such properties can be formulated, for instance, as assertions in the code.

The motivation for this form of specification is that the properties are usable through test case generation for bug hunting and also by formal verification tools. In test case generation, they can be used to guide the construction of test vectors and as an oracle to determine whether a test passes or fails. As mentioned, the target application for our work is acceptance and regression testing. So, we want to find test cases that check whether the requirement expressed by a property has been implemented at all—in essence finding constructive proof that the property is not vacuously satisfied simply because a feature has been left out or has been removed. We therefore consider a property $\mathbf{G}(\Phi)$ to be 'covered' if we can satisfy Φ non-vacuously at some point in the execution of the test.

The formula Φ of a property is a logical implication of the general form:

$$\varphi(\boldsymbol{s}_k, \boldsymbol{i}_k, \ldots, \boldsymbol{s}_{k+J_\varphi-1}, \boldsymbol{i}_{k+J_\varphi-1}) \Rightarrow \psi(\boldsymbol{s}_{k+J_\varphi}).$$

The antecedent φ describes the test goal in the form of an assumption about the values of the state variables and input variables. The consequent ψ is a formula over the state variables. It defines the test outcome to be checked. Formally, $\boldsymbol{s}_k, \ldots, \boldsymbol{s}_{k+J_\varphi}$ are all vectors of state variables of equal, but indeterminate length. Likewise, $\boldsymbol{i}_k, \ldots, \boldsymbol{i}_{k+J_\varphi-1}$ are all vectors of input variables. The indices $k, \ldots k + J_\varphi$ name the

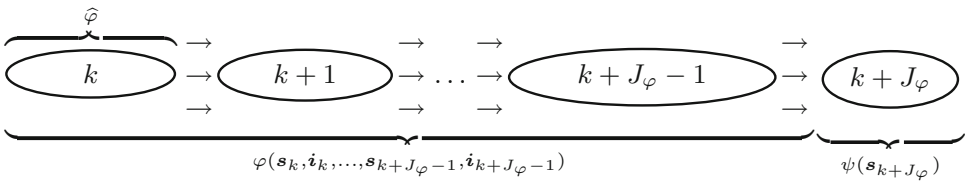


Fig. 3 Structure of properties

successive steps in an execution. Note that Φ characterizes a set of finite sub-executions of length J_φ.

Only the antecedent φ stating the test goal of a property such as Φ above is needed for test vector generation. Given a collection of properties, we write Π for the set of all such antecedents.

For example, for property p_1 in our running example, we have

$$\varphi = \big((mode = ON) \wedge (speed = 1) \wedge dec\big)$$

and $\psi = (speed = 1)$ where *mode* and *speed* are state variables and *dec* is an input variable.

An execution $(S_0 \xrightarrow{I_0} S_1 \xrightarrow{I_1} S_2 \rightarrow \cdots)$ *covers* a property iff it contains a sub-execution $(S_k \xrightarrow{I_k} \cdots S_{k+J_\varphi-1} \xrightarrow{I_{k+J_\varphi-1}} S_{k+J_\varphi})$ that satisfies φ, i.e.,

$$\exists k \geq 0 : \exists S_{k+J_\varphi} : \varphi(S_k, I_k, \ldots, S_{k+J_\varphi-1}, I_{k+J_\varphi-1}) \wedge \bigwedge_{k+1 \leq m \leq k+J_\varphi} T(S_{m-1}, I_{m-1}, S_m).$$

We call the formula describing the union of all first states S_k in any sub-execution satisfying φ the *trigger* $\widehat{\varphi}$ of the property. Figure 3 illustrates this diagrammatically; a property extends over a finite segment of a set of executions, and the trigger for the property is the set of start states of this segment. For example, property p_1 in our running example has the trigger $\big((mode = ON) \wedge (speed = 1)\big)$.

We assume that property antecedents are non-overlapping, i.e. the sub-executions satisfying the antecedents do not share any edges. Our minimality results only apply to such specifications. Detecting overlappings is a hard problem [8] that goes beyond the scope of this article.

Test cases. A *test case* (of length n) is an input sequence $\langle I_0, \ldots, I_{n-1}\rangle$ and generates an execution $(S_0 \xrightarrow{I_0} \cdots \xrightarrow{I_{n-1}} S_n)$ with n transitions—i.e. an execution of length n. (An execution of length 0 is just an initial state.) A test case *covers* a property p exactly when its execution covers the property.

3 Chaining test cases

The problem. We are given a program $(\Sigma, \Upsilon, T, Init)$, properties $\mathcal{P}$, and a set of final states $Final \subseteq \Sigma$. A *test case chain* χ is a test case $\langle I_0, \ldots, I_{n-1}\rangle$ that covers all properties in $\mathcal{P}$, i.e., its execution $(S_0 \xrightarrow{I_0} \cdots \xrightarrow{I_{n-1}} S_n)$ starts in $Init(S_0)$, ends in $Final(S_n)$, and covers all properties in $\mathcal{P}$. A *minimal test case chain* is a test case chain of minimal length. The final states *Final* are used to ensure the test execution ends in a desired state, e.g. 'engines off' or 'gear locked in park mode'.

Our approach. We now describe our basic algorithm, which has three steps:

1. *Abstraction.* We construct a *property K-reachability graph* of the system. This is a weighted, directed graph with nodes representing the properties and edges labelled with the number of states through which execution must pass, up to length K, between the properties.
2. *Optimization.* We determine the shortest path that covers all properties in the abstraction.
3. *Concretization.* Finally, we compute the corresponding concrete test case chain along the abstract path.

This algorithm is given as Algorithm 1.

Algorithm 1: Compute test case chain

Input: program $(\Sigma, \Upsilon, T, Init)$, properties $\mathcal{P}$, formulas *Init*, *Final*, reachability bound K
Output: test case chain $\chi = \langle I_0, \ldots, I_N\rangle$
1 $G = BuildPropKReachGraph(\Pi, Init, Final, T, K)$
2 $\pi = GetShortestPath(G, Init, Final)$
3 $\chi = GetChain(G, \pi, T)$
4 **return** χ

In Sect. 3.4 we discuss the conditions under which we obtain the *minimal* test case chain.

3.1 Abstraction: property K-reachability graph

The *property K-reachability graph* is an abstraction of the program by a weighted, directed graph (V, E, W), with

- vertices $V = \Pi \cup \{Init, Final\}$, all defining property antecedents, including formulas describing the sets *Init* and *Final*,
- edges $E \subseteq E_{target} \subset V \times V$, as explained below, and
- an edge labelling $W : E \rightarrow \mathbb{N}$ assigning to each $(\varphi, \varphi') \in E$ the minimal number of transitions bounded by K needed to reach some state satisfying the property trigger $\widehat{\varphi'}$ by extending any subexecution satisfying φ according to the program's transition relation T.

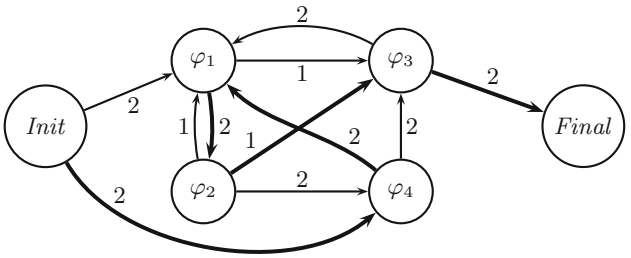


Fig. 4 Test case chaining: property K-reachability graph (for $K = 2$) and minimal test case chain of length $n = 9$ (*bold edges*) for our example (Fig. 2)

Figure 4 shows the property 2-reachability graph for our example.

Graph construction. The graph is constructed by the function *BuildPropKReachGraph*$(\Pi, Init, Final, T, K)$ (see Algorithm 2). The main work is done by the function *GetKreachEdges*$((V, E, W), T, E_{target}, k)$ (see Algorithm 10 in Sect. 5 for details), which computes the subset of edges E_k that have weight k in the set of interesting edges E_{target} which initially contains all pairwise links between the nodes φ_j, links from *Init* to all nodes φ_j, and from every φ_j to *Final*. The constructed graph contains an edge (φ, φ') with weight k iff, for the two properties with antecedents φ and φ', $k \leq K$ is the minimal number of transitions needed to extend a sub-execution satisfying φ to reach a state in $\widehat{\varphi'}$. We stop the construction of the graph if a path has been found (line 5). *ExistsPath* is explained below. If we fail to find a path before reaching a given reachability bound K, or there is no path although the graph contains all edges in E_{target}, then we abort (line 6).

Algorithm 2: *BuildPropKReachGraph*

Input: property antecedents Π, formulas *Init*, *Final*, transition function T, reachability bound K
Output: weighted, directed graph (V, E, W)

1 $V \leftarrow \Pi \cup \{Init, Final\}$
2 $E \leftarrow \emptyset, W \leftarrow \emptyset$
3 $E_{target} \leftarrow \left(\bigcup_{\varphi_j \in \Pi} \{(Init, \varphi_j), (\varphi_j, Final)\}\right) \cup \{(\varphi_j, \varphi_k) \mid \varphi_j, \varphi_k \in \Pi, j \neq k\}$
4 $k \leftarrow 0$
5 **while** $\neg$*ExistsPath*$((V, E, W), Init, Final)$ **do**
6 **if** $k > K \vee E_{target} = \emptyset$ **then abort** 'no chain found for given bound K'
7 **let** $E_k = GetKreachEdges((V, E, W), T, E_{target}, k)$
8 $E \leftarrow E \cup E_k, E_{target} \leftarrow E_{target} \setminus E_k$
9 **for all** $e \in E_k$ **do** $W \leftarrow W \cup \{e \mapsto k\}$
10 $k \leftarrow k + 1$
11 **return** (V, E, W)

Existence of a covering path. Algorithm 2 requires to check for the existence of a covering path (function *ExistsPath*) in each iteration. We formulate the existence of a covering path as a reachability problem in a directed graph:

Lemma 1 *Let* (V, E, W) *be a property* K*-reachability graph. Then, there is a covering path from Init to Final iff*

(1) *all vertices are reachable from Init*,
(2) *Final is reachable from all vertices, and*
(3) *for all pairs of vertices* $(v_1, v_2) \in (V \setminus \{Init, Final\})^2$,
   (a) v_2 *is reachable from* v_1 *or* (b) v_1 *is reachable from* v_2.

Proof In the transitive closure (V, E', W') of (V, E, W), v_2 is reachable from v_1 iff there exists an edge $(v_1, v_2) \in E'$.

($\Longrightarrow$): conditions (1) and (2) are obviously necessary. Let us assume that we have a covering path π and there are vertices (v_1, v_2) which neither satisfy (3a) nor (3b). Then neither $\langle v_1, \ldots, v_2 \rangle$ nor $\langle v_2, \ldots, v_1 \rangle$ can be a subpath of π, which contradicts the fact that π is a covering path.

($\Longleftarrow$): Any vertex is reachable from *Init* (1), so let us choose v_1. From v_1 we can reach another vertex v_2 (3a), or, at least, v_1 is reachable from another vertex v_2 (3b), but in the latter case, since v_2 is reachable from *Init*, we can go first to v_2 and then to v_1. Induction step: let us assume we have a path $\langle Init, v_1, \ldots, v_k \rangle$. If there is a vertex v' that is reachable from v_k (3a), we add it to our current path π. If v' is unreachable from v_k, then by (3b), v_k must be reachable from v', and there is a $v_i, i < k$ in $\pi = \langle Init, \ldots, v_k \rangle$ from which it is reachable and in this case we obtain the path $\langle Init, \ldots, v_i, v', v_{i+1}, \ldots, v_k \rangle$; if there is no such v_i then, at last by (1), v' is reachable from *Init*, so we can construct the path $\langle Init, v', \ldots, v_k \rangle$. *Final* is reachable from any vertex (2); thus, we can complete the covering path as soon as all other vertices have been covered. □

Reachability of a vertex from another vertex can be checked in constant time on the transitive closure of the graph. Hence, the overall existence check has complexity $\mathcal{O}(|V|^3)$. The algorithm is listed in Algorithm 3.

Algorithm 3: *ExistsPath*

Input: transitive closure of directed graph (V, E)
Output: existence of a covering path π

1 $v \leftarrow chooseFrom(V); V \leftarrow V \setminus \{v\}; \pi \leftarrow \langle v \rangle$
2 **while** $V \neq \emptyset$ **do**
3 $v \leftarrow chooseFrom(V); V \leftarrow V \setminus \{v\}; v' \leftarrow lastElement(\pi)$
4 **if** $(v', v) \in E$ **then** $\pi \leftarrow append(\pi, v)$
5 **else if** $(v, v') \in E$ **then**
6 **while** $(v', v) \notin E$ **do** $v' \leftarrow previousElement(\pi, v')$
7 $\pi \leftarrow insertAfter(\pi, v, v')$
8 **else return** *false* //no path found
9 **return** *true*

3.2 Optimization: shortest path computation

The next step is to compute the shortest path (function *GetShortestPath* in Algorithm 1) covering all nodes in the prop-

erty K-reachability graph. Such a path is not necessarily Hamiltonian; revisiting nodes is allowed. However, we can compute the transitive closure of the graph using the Floyd–Warshall algorithm [9] (which preserves minimality), and then compute a Hamiltonian path from *Init* to *Final*. If we do not have a Hamiltonian path solver, we can add an edge from *Final* to *Init* and pass the problem to an *asymmetric travelling salesman problem* (ATSP) solver (referred to as *SolveATSP* in the sequel) that gives us the shortest circuit that visits all vertices exactly once. We cut this circuit between *Final* and *Init* to obtain the shortest path π.

Lemma 2 (Minimum covering path) *Let* (V, E', W') *be the transitive closure of a property* K*-reachability graph* (V, E, W)*, and suppose* $Init, Final \in V$*. Then, SolveATSP* $(V, E' \cup \{(Final, Init)\}, W' \cup \{(Final, Init) \mapsto 1\})$ *returns a Hamiltonian circuit* $\langle v_0, \ldots, v_{|V|-1}\rangle$ *such that* $\pi = \langle v_i, \ldots, v_{|V|-1}, v_0, \ldots, v_{i-1}\rangle$ *with* $v_i = Init$ *and* $v_{i-1} = Final$ *is a minimum covering path from Init to Final in* (V, E', W').

Proof Suppose (V, E, W) has a circuit $\langle \ldots, v, v', v, v'', \ldots\rangle$ that covers all vertices but is non-Hamiltonian—because the vertex v, say, is visited twice. Then in the transitive closure (V, E', W') we can bypass v because v'' is now directly reachable from v'. Hence, we obtain a Hamiltonian circuit $\langle \ldots, v, v', v'', \ldots\rangle$. We extend the graph to $(V, E' \cup \{(Final, Init)\}, W' \cup \{(Final, Init) \mapsto 1\})$, i.e. we add the edge $(Final, Init)$ with weight 1.

Any Hamiltonian circuit $\langle v_0, \ldots, v_{|V|-1}\rangle$ returned by *SolveATSP* for the extended graph must contain the edge $(v_{(i-1) \mod |V|}, v_i) = (Final, Init)$, because $(Final, Init)$ is the only (and hence the cheapest) edge for reaching *Init* from *Final*. Hence, $\langle v_i, \ldots, v_{|V|-1}, v_0, \ldots, v_{i-1}\rangle$, i.e. the Hamiltonian circuit cut between *Final* and *Init*, is a Hamiltonian path of minimum length, because the transitive closure preserves optimality ($W(v_1, v_2) + W(v_2, v_3) = W(v_1, v_3)$). □

For our example, the shortest path has length 9, given as bold edges in Fig. 4.

3.3 Concretization: computing the test case chain

Once we have found a minimum covering path π in the property K-reachability graph abstraction, we have to compute the inputs corresponding to it in the program. This is done by the function *CheckPath*(π, T, W) which takes an abstract path $\pi = \langle \varphi_1, \ldots, \varphi_{|V|}\rangle$ and returns the input sequence $\langle I_0, \ldots, I_n\rangle$ corresponding to a concrete execution with the reachability distances between each $(\varphi_j, \varphi_{j+1}) \in \pi$ given by the edge weights $W(\varphi_j, \varphi_{j+1})$. Typically, *CheckPath* involves constraint solving; we will discuss our implementation in Sect. 5. Hence, *GetChain* in Algorithm 1 corresponds to a call to *CheckPath*(π, T, W) and returning the obtained input sequence.

For our example, we obtain, for instance, the sequence $\langle$*gas*, *acc*, *button*, *dec*, *dec*, *gas*, *dec*, *brake*, *button*$\rangle$ corresponding to the bold edges in Fig. 2.

3.4 Optimality

Since the (non-)existence or the optimality of a covering path in the K-reachability graph does not imply the (non-) existence or the optimality of a chain in the program, the success of this procedure can only be guaranteed under certain conditions, which we now discuss.

Lemma 3 (Single-state property triggers) *The test case chain computed by Algorithm 1 is minimal provided that*

(1) *the program and the properties admit a test case chain,*
(2) *all triggers* $\widehat{\varphi}$ *of properties in* $\mathcal{P}$ *are singleton sets, and*
(3) *the test case chain* χ *computed by Algorithm 1 visits each property once.*

Proof If [assumption (3)] each property is visited once, it is guaranteed that the covering path in the K-reachability graph contains only edges that correspond to concrete paths of minimal length in the program. Otherwise, for a subpath $(\varphi, \varphi', \varphi, \varphi'')$, there might exist an edge (φ', φ'') with $W(\varphi', \varphi'') < W(\varphi', \varphi) + W(\varphi, \varphi'')$ that is only discovered for higher values of K. Due to assumption (2), the concretization is guaranteed to succeed. Hence, the test case chain χ is optimal for the program and the given properties, unless [assumption (1)] they do not admit a test case chain at all. □

For finite state systems, there is an upper bound for K, the reachability diameter [10] beyond which we will not discover shorter pairwise links.

Definition 1 (*Reachability diameter*) The reachability diameter N of a program $(\Sigma, \Upsilon, T, Init)$ is the maximum (finite) length of an execution in the set of shortest executions between any pair of states $S_i, S_j \in \Sigma$.

Theorem 1 (Minimal test case chain) *Let* N *be the reachability diameter of the program; then there is a* $K \leq N$ *such that, under the preconditions* (1) *and* (2) *of Lemma 3, the test case chain* χ *computed by Algorithm 1 is minimal.*

Proof For $K = N$, it is guaranteed that the minimal covering path in the K-reachability graph contains only edges of minimal length, and hence the chain is optimal w.r.t. the program (even if properties are revisited).

Computing N exactly is as difficult as the model checking problem itself. There are methods for estimating bounds [10,11], but these are often overly conservative. Heuristic

approaches to algorithmically choosing an appropriate K might be worthwhile investigating, but go beyond the scope of this article. In practice, therefore, we propose to manually stop the procedure if a chain of acceptable length is found—i.e. in our implementation we do not estimate the reachability diameter, but use a user-supplied bound.

Since K is not a bound on the length of the chain but a bound on the distance between two properties, one can hope that in most cases, we can find a minimal chain using a K that is smaller than the length of the minimal chain itself. This is confirmed by our experiments.

4 Generalizations

We will now generalize our algorithm in three ways:

1. *Multi-state property triggers.* Dropping the assumption that triggers are single state may make the concretization phase fail. Under certain restrictions, we will still find a test case chain if one exists, but we lose minimality.
2. Without these restrictions, we might even lose completeness, i.e., the guarantee to find a chain if one exists. We propose two methods to *ensure completeness* under these circumstances: (1) an abstraction refinement that can be used with any ATSP solver, and (2) a method based on restricting the optimization problem using path constraints that requires a more general solver, e.g. an Answer Set Programming (ASP) solver.
3. *Multiple chains.* Dropping the assumption about the existence of a single chain raises the problem of how to generate multiple chains.

4.1 Multi-state property triggers

In practice, many properties are multi-state, i.e. preconditions (2) of Lemma 3 is not met. In this case, the abstract covering path might be infeasible in the concrete program, and hence the naive concretization of Sect. 3.3 might fail. We have to extend the concretization step to fix such broken chains.

Example 1 (Broken chain) Let us consider the following broken chain in our example with the properties:

$$p_1\colon\ \mathbf{G}\big(mode = OFF \wedge \neg enable \wedge button \Rightarrow \mathbf{X}\, enable\big)$$
$$p_2\colon\ \mathbf{G}\big(mode = ON \wedge brake \Rightarrow \mathbf{X}(mode = DIS)\big)$$

with $Init = Final = \{mode = OFF \wedge speed = 0 \wedge \neg enable\}$.

We obtain a shortest covering path $\langle Init, \varphi_1, \varphi_2, Final\rangle$ in the abstraction with weights $W(Init, \varphi_1) = 0$, $W(\varphi_1, \varphi_2) = 1$, and $W(\varphi_2, Final) = 2$. However, Fig. 2 tells us that the path $\langle Init, \varphi_1, \varphi_2\rangle$ is not feasible in a single transition, but requires two transitions, as illustrated in Fig. 5.

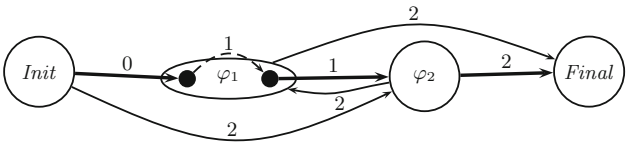


Fig. 5 Broken chain: the path $\langle Init, \varphi_1, \varphi_2\rangle$ is not feasible in a single transition, but requires two transitions

A broken chain contains an infeasible subpath *failed_path* $= \langle \varphi_1, \ldots, \varphi_k\rangle$ of the abstract path π that involves at least three vertices, such as $\langle Init, \varphi_1, \varphi_2\rangle$ in our example above. We extend the concretization step (*GetChain*) with a chain repair capability (see Algorithm 4). The function *RepairPath* as shown in Algorithm 11 iteratively repairs broken chains by incrementing the weights associated with the edges of *failed_path* and checking the feasibility of this 'stretched' path. We give more details about our implementation in Sect. 5.

Algorithm 4: *GetChain* with chain repair

Input: weighted, directed graph (V, E, W), path π, transition relation T, reachability bound K
Output: test case chain $\chi = \langle I_0, \ldots, I_N\rangle$

```
(feasible, χ, failed_path) ← CheckPath(π, T, W)
if feasible then return χ
else
    (succeeded, W, _) ← RepairPath(failed_path, T, W)
    if ¬succeeded then abort 'no chain found for given bound K'
    (_, χ, _) ← CheckPath(π, T, W)
    return χ
```

Example 2 (Repaired chain) For the broken chain in our previous example, we will check whether $\langle Init, \varphi_1, \varphi_2\rangle$ is feasible with $W(\varphi_1, \varphi_2)$ incremented by one. This makes the path feasible and we obtain the chain $\chi = \langle button, gas, brake, button\rangle$.

Completeness. The chain repair succeeds if the given path π admits a chain in the concrete program. In particular, this holds when the states in each property trigger are strongly connected:

Theorem 2 (Multi-state strongly connected property) *If for each property trigger $\widehat{\varphi}$ the states are strongly connected and there exists a test case chain, then Algorithm 1 (with Algorithm 4) will find it.*

In practice, many reactive systems are, apart from an initialization phase, strongly connected—but, as stressed above, the test case chain might not be minimal.

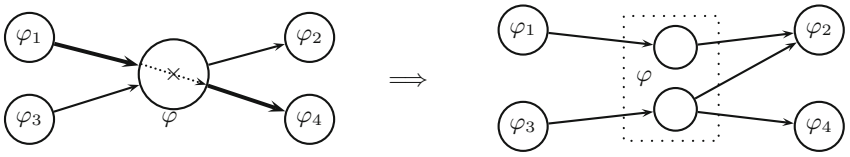


Fig. 6 Abstraction refinement for a failed path $\langle \varphi_1, \varphi, \varphi_4 \rangle$ (*bold arrows*)

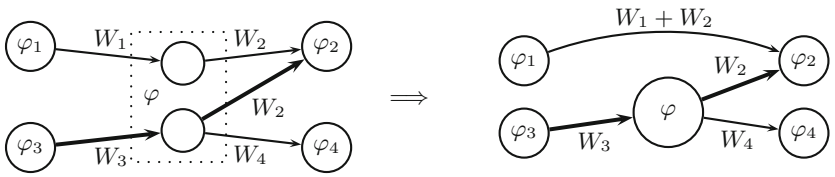


Fig. 7 Collapsing the property refinement group (*box*) in the refined abstraction to a TSP problem w.r.t. a solution path (*bold arrows*)

Algorithm 5: *GetChain* with abstraction refinement

Input: weighted, directed graph (V, E, W), path π, transition relation T, reachability bound K
Output: test case chain $\chi = \langle I_0, \ldots, I_N \rangle$

1 $G \leftarrow \{\{v\} \mid v \in V\}$ //property refinement groups
2 **while** *true* **do**
3 $(feasible, \chi, failed\_path) \leftarrow CheckPath(\pi, T, W)$
4 **if** *feasible* **then return** χ
5 $(succeeded, W', failed\_path) \leftarrow RepairPath(failed\_path, T, W)$
6 **if** *succeeded* **then**
7 $(\_, \chi, \_) \leftarrow CheckPath(\pi, T, W')$
8 **return** χ
9 $(V, E, W, G) \leftarrow refine(V, E, W, failed\_path, G)$
10 $\pi \leftarrow GetCoveringPath(V, E, G)$
11 **if** $\pi = \langle\rangle$ **then abort** 'no chain found for given bound K'
12 $(V, E, W) \leftarrow collapse(V, E, W, \pi, G)$
13 $\pi \leftarrow GetShortestPath(V, E, W)$

4.2 Ensuring completeness

If the shortest path π in the abstraction does not admit a chain in the concrete program, Algorithm 1 using *GetChain* with chain repair (Algorithm 4) will fail to find a test case chain even though one exists, i.e., it is not complete.

Example 3 (Chain repair fails) In Fig. 5, we have found the shortest abstract path $\langle Init, \varphi_1, \varphi_2, Final \rangle$. Now assume that the right state in φ_1 is not reachable from the left state. Then the chain repair fails. In this case, there might still be a (non-)minimal path in the abstraction that admits a chain: in our example in Fig. 5, assuming that the left state in φ_1 is reachable from *Init* via φ_2 and *Final* is reachable from the left state in φ_1, we have the feasible path $\langle Init, \varphi_2, \varphi_1, Final \rangle$.

Abstraction refinement. To obtain completeness in this situation, we propose the following abstraction refinement method sketched in Algorithm 5. Suppose the chain repair of a covering path π failed with $failed\_path = \langle \varphi_1, \varphi, \varphi_4 \rangle$ ($succeeded = false$ in line 5).

1. We refine the graph by splitting vertex φ in *failed_path* as illustrated in Fig. 6 that rules out the infeasible subpath, as typically done by abstract refinement algorithms (represented by function *refine* in line 9). We call the vertices obtained from such splittings that belong to the same property a *property refinement group* (subsets of G).
2. Then we adapt the Algorithm 3 that we use for checking the existence of a covering path to finding a (non-minimal) covering path from *Init* to *Final*, taking into account that a covering path needs to cover only one vertex for each property refinement group (*GetCoveringPath* called in line 10 of Algorithm 5).
3. A solution π obtained that way might be far from optimal, so we exploit the TSP solver to give us a better solution π'. However, the refined graph does not encode the desired TSP problem because it is sufficient to cover only one vertex for each property refinement group. Hence, given a path π, we transform the graph by collapsing each property refinement group with respect to π as illustrated by Fig. 7 (represented by function *collapse* in line 12 of Algorithm 5). The obtained graph is handed over to the TSP solver (line 13). Note that the transformations do not preserve optimality, because, e.g. in Fig. 7, the edge (φ_1, φ_2) would cover φ in a concrete path, but not in the transformed, refined abstract graph.
4. We try to compute a concrete test case chain for the covering path (lines 3–8). If this fails, we iterate the refinement process.

In each iteration (line 2) of the abstraction refinement algorithm, a node in the graph is split such that a concrete spurious transition is removed from the abstraction, i.e. the transition system structure of the program inside the property antecedents is made more explicit in the abstraction. Provided the existence of a test case chain, since there is only a finite number of transitions, the abstraction refinement will eventually terminate and a covering path will be found that can be concretized to a test case chain.

Example 4 (Abstraction refinement) Assume, as in the previous example, that the right state in φ_1 in Fig. 5 is not reachable from the left state. Then the abstraction refinement will split

φ_1 into two vertices. Suppose that *GetCoveringPath* returns the covering path $\pi = \langle Init, \varphi_2, \varphi_1, \varphi_2, Final\rangle$.[2] Then collapsing the two nodes belonging to φ_1 w.r.t. π will remove the edge from *Init* to φ_1. The TSP solver will optimize π and find the shorter path $\langle Init, \varphi_2, \varphi_1, Final\rangle$.

Path constraints. The fundamental problem about a failed path is that it represents information about at least two edges that we cannot encode as an equivalent TSP. We would need a TSP solver that can deal with side conditions like the following: the solution must not contain vertices v_1, v_2, v_3 in this particular order for any infeasible subpath $\langle v_1, v_2, v_3\rangle$ in *failed_path*. Similar difficulties arise concerning *minimality*: here, we would have to add 'path weights' that penalize a solution if it contains a certain path. To address this problem, we can opt using answer set programming (ASP) solvers (e.g. [12]), which are far less efficient in solving TSPs, but allow us to specify arbitrary side conditions.

Example 5 (Path constraints) Consider the graph in Fig. 5. We can encode the TSP problem in ASP as follows (cf. [12]):

```
  V(I,phi1,phi2,F).

E(I,phi1). weight(I,phi1,0).
E(I,phi2). weight(I,phi2,2).
E(phi1,phi2). weight(phi1,phi2,1).
E(phi1,F). weight(phi1,F,2).
E(phi2,phi1). weight(phi2,phi1,2).
E(phi2,F). weight(phi2,F,2).

{ cycle(X,Y) : E(X,Y) } I :- V(X).
{ cycle(X,Y) : E(X,Y) } I :- V(Y).

 reached(Y) :- cycle(I,Y).
 reached(Y) :- cycle(X,Y), reached(X).
:- V(Y), not reached(Y).

#minimize [ cycle(X,Y) : weight(X,Y,C) = C ].
```

Assume, again, that the right state in φ_1 in Fig. 5 is not reachable from the left state, so that we obtain *failed_path* $= \langle Init, \varphi_1, \varphi_2\rangle$. Then we can exclude *failed_path* by adding

```
  twopath(X,Y,Z) :- cycle(X,Y), cycle(Y,Z).
 -twopath(I,phi1,phi2).
```

to the ASP problem. The ASP solver will return the shortest covering path that does not contain *failed_path*, i.e. $\langle Init, \varphi_2, \varphi_1, Final\rangle$.

To use path constraints, lines 9–13 in Algorithm 5 are replaced by a call to the ASP solver with the path constraints obtained from *failed_path* (followed by the check in line 11 that a path was actually found).

[2] It will actually return the better result for this particular example.

4.3 Multiple chains

We can relax our problem to systems and properties that do not admit single chains. A system does not admit a single chain if two of the given properties never become both true in any execution of the system. This means that we drop condition (3) of Lemma 1. For example, we have three distinct states S, S', S'' and we want to cover the two transitions (S', I', S) and (S'', I'', S), where neither S' is reachable from S nor S'' is reachable from S. In this case, we require two chains to cover both transitions. Note that such systems still have to satisfy conditions (1) and (2) of Lemma 1 to guarantee the existence of multiple chains.

We can detect that a system does not admit a single chain if

1. the N-reachability property graph has no chain (where N is the reachability diameter of the system), or
2. the chain repair or abstraction refinement process fails.

We use Lemma 1 to devise an algorithm for computing a partition $\{P_1, \ldots, P_n\}$ of $\mathcal{P}$ (function *Partition*, Algorithm 7) and apply Algorithm 1 for each P_i. If the chain repairing fails for a P_i, we compute a partition for the refined property graph. The overall algorithm is sketched in Algorithm 6.

Finding the smallest partition is equivalent to the problem of finding a vertex colouring with minimal chromatic number (NP-hard problem) w.r.t. the conflict relation R, i.e. inconsistency regarding condition (3) of Lemma 1. Finally, the remaining (non-conflicting) vertices are added to some element of the partition, and *Init* and *Final* are added to all partitions (lines 5 and 6 of Algorithm 7).

Remark 1 Given a bound K, our algorithm stops at the smallest k such that the k-reachability graph has a covering path. Theorems 1 and 2, and the methods proposed in Sect. 4.2 guarantee completeness for this k-reachability graph. However, Sect. 4.2 does not guarantee completeness w.r.t. K,

Algorithm 6: Compute multiple test case chains

Input: program $(\Sigma, \Upsilon, T, Init)$, properties $\mathcal{P}$, formulas *Init*, *Final*, reachability bound K
Output: test case chains $\chi = \langle I_0, \ldots, I_N\rangle$

```
1  G = BuildPropKReachGraph(Π, Init, Final, T, K)
2  S = Partition(G)
3  for all G ∈ S do
4      π = GetShortestPath(G, Init, Final)
5      try
6          χ = GetChain(G, π, T)
7          output χ
8      catch(abort)
9          S ← (S ∪ Partition(G)) \ {G}
```

Algorithm 7: *Partition*

Input: directed, weighted graph (V, E, W)
Output: partition S of V

if (V, E, W) *does not satisfy conditions (1), (2) of Lem. 1* **then**
  abort 'no chain found for given bound K'
$R =$ set of pairs $(v_i, v_j) \in V$ that do not satisfy condition (3) of Lem. 1.
$S = \bigcup_{1 \le i \le n} \{P_i\}$ partition of $V_R = \bigcup_{(v,v') \in R} \{v, v'\}$ with minimal n s.t. $\forall P \in S, \forall (v, v') \in R : \{v, v'\} \not\subseteq P_i$.
for all $P \in S$ **do** $P \leftarrow P \cup \{Init, Final\}$
choose $P \in S$: $P \leftarrow P \cup (V \setminus V_R)$
return S

Algorithm 8: *CheckPath*

Input: path π, transition relation T, weights W
Output: whether π is *feasible*, *inputs* associated to π if feasible, *failed_path* $\subseteq \pi$ if infeasible

$inputs \leftarrow \langle\rangle$
$failed\_path \leftarrow \langle\rangle$
$(feasible, assignment, unsat\_core) = SAT(BuildPath(\pi, T, W))$
if *feasible* **then**
  let $(S_0, I_0, S_1, I_1, \ldots, S_K, I_K) = assignment$
  $inputs \leftarrow \langle I_0, \ldots, I_N \rangle$
else
  $failed\_path \leftarrow getFailedPath(unsat\_core, \pi)$
return $(feasible, inputs, failed\_path)$

Algorithm 9: *BuildPath*

Input: path π, transition relation T, weights W
Output: path formula Φ

return $BuildPathRec(\pi, 0, true)$
function $BuildPathRec(\pi, k, \Phi)$
  if $lengthOf(\pi) = 1$ **then**
    let $\varphi = \pi$
    return $\Phi \wedge \widehat{\varphi}(\boldsymbol{s}_k)$
  else
    let $\langle v, \pi_{tail} \rangle = \pi$
    let $\langle v', \_ \rangle = \pi_{tail}$
    let $k_{end} = k + W(v, v')$
    let $\varphi = v$
    return $\Phi \wedge \varphi(\boldsymbol{s}_k, \boldsymbol{i}_k, \ldots, \boldsymbol{s}_{k+J_\varphi}, \boldsymbol{i}_{k+J_\varphi}) \wedge \bigwedge_{k+1 \le j \le k_{end}} T(\boldsymbol{s}_{j-1}, \boldsymbol{i}_{j-1}, \boldsymbol{s}_j) \wedge BuildPathRec(\pi_{tail}, k_{end}, \Phi)$

Algorithm 10: *GetKreachEdges*

Input: weighted, directed graph (V, E, W), transition relation T, edges to be considered E_{target}, number of transitions k
Output: k-reach edges $E_k \subseteq E_{target}$

$from\_to \leftarrow E_{target}$
$E_k \leftarrow \emptyset$
$(sat, assignment) \leftarrow checkKreach(from\_to, T, k)$
while *sat* **do**
  let $(S_0, I_0, S_1, I_1, \ldots, S_{J_\varphi+k}, I_{J_\varphi+k}) = assignment$
  for all $v, v' \in V$: $\varphi = v, \varphi' = v' : \varphi(S_0, I_0, \ldots, S_{J_\varphi-1}, I_{J_\varphi-1}) \wedge \varphi'(S_{J_\varphi+k})$ **do**
    $E_k \leftarrow E_k \cup \{(v, v')\}$
    $from\_to \leftarrow from\_to \setminus \{(v, v')\}$
  $(sat, assignment, \_) \leftarrow checkKreach(from\_to, T, k)$
return E_k

Algorithm 11: *RepairPath* by concrete chaining

Input: *failed_path*, transition relation T, weights W, reachability bound K
Output: updated weights W, failed path if repair fails

$W' \leftarrow W$
$\sigma \leftarrow FirstElement(failed\_path)$
for $j = 1$ ***to*** $n - 2$ **do**
  $e = (\varphi_j, \varphi_{j+1})$
  $feasible \leftarrow false$
  while $\neg feasible$ **do**
    $(feasible, assignment, \_) \leftarrow CheckPath(\langle \sigma, \varphi_{j+1} \rangle, T, W)$
    if $\neg feasible$ **then** $W(e) \leftarrow W(e) + 1$
    else
      let $\langle S_0, \ldots \rangle = assignment$
      $\sigma \leftarrow S_0$
    if $W(e) > K$ **then return**
    $RepairPath'(failed\_path, T, W', K)$
return $(true, W, \langle\rangle)$

Algorithm 12: *RepairPath′*

Input: *failed_path*, transition relation T, weights W, reachability bound K
Output: updated weights W, failed path if repair fails

$\langle \varphi_0, \ldots, \varphi_{n-1} \rangle = failed\_path$
for $j = 1$ ***to*** $n - 2$ **do**
  $e = (\varphi_j, \varphi_{j+1})$
  $feasible \leftarrow false$
  while $\neg feasible$ **do**
    $(feasible, \_, \_) \leftarrow CheckPath(\langle \varphi_0, \ldots, \varphi_{j+1} \rangle, T, W)$
    if $\neg feasible$ **then** $W(e) \leftarrow W(e) + 1$
    if $W(e) > K$ **then return** $(false, W, \langle \varphi_{j-1}, \varphi_j, \varphi_{j+1} \rangle)$
return $(true, W, \langle\rangle)$

because for $k < K$ there will be additional covering paths in the K-reachability graph that are not considered by the methods in Sect. 4.2. To ensure completeness w.r.t. K, one would have to extend the k-reachability graph to a K-reachability graph before applying Sect. 4.2. Otherwise, we might generate multiple chains where we could have found a single chain in the K-reachability graph.

5 Test-case generation with bounded model checking

The previous sections abstract from the actual back-end implementation of the functions *GetKreachEdges*, *CheckPath*, and *RepairPath*. In this work, we use bounded model checking to provide an efficient implementation. Alterna-

tive instantiations could be based, for example, on symbolic execution.

BMC-based test case generation. Bounded model checking (BMC) [13] can be used to check the existence of an execution of increasing length K from ϕ to ϕ'. This check is performed by deciding the satisfiability of the following formula using a SAT solver:

$$\phi(\boldsymbol{s}_0) \wedge \bigwedge_{1 \le k \le K} T(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k) \wedge \phi'(\boldsymbol{s}_K) \tag{1}$$

If the SAT solver returns the answer *satisfiable*, it also provides a satisfying assignment $(S_0, I_0, S_1, I_1, \ldots, S_{K-1}, I_{K-1}, S_K)$. The satisfying assignment represents one possible execution from ϕ to ϕ' and identifies the corresponding input sequence $\langle I_0, \ldots, I_{K-1} \rangle$.

Instantiation. For implementing Algorithm 1 with chain repair (Algorithm 4), we have to provide the functions *CheckPath*, *GetKreachEdges*, and *RepairPath*.

We consider a SAT solver to be a function $SAT : \phi \mapsto (sat, assignment, unsat\_core)$ where *assignment* contains a satisfying assignment if ϕ is sat and otherwise *unsat_core* is a minimal formula such that $\phi \Rightarrow unsat\_core$ and $\neg unsat\_core \Rightarrow \neg\phi$.

Then, *CheckPath* is defined as in Algorithm 8 where *BuildPath* constructs the BMC formula for a given path, and *getFailedPath* converts an *unsat_core* into a path, the implementation of which is SAT/SMT solver specific. For example, in MINISAT [14] one can *assume* the property antecedents φ instead of adding them to the formula. If the formula Algorithm 9 is unsatisfiable, MINISAT returns the list of those antecedents that contributed to the final conflict.

GetKreachEdges is given as Algorithm 10, where the function $checkKreach(\pi, T, k)$ that is used for enumerating K-reachability edges is implemented by checking the satisfiability of the following formula for each φ:

$$\begin{aligned} &\varphi(\boldsymbol{s}_0, \boldsymbol{i}_0, \ldots, \boldsymbol{s}_{J_\varphi - 1}, \boldsymbol{i}_{J_\varphi - 1}) \\ &\wedge \bigwedge\nolimits_{1 \le j \le J_\varphi + k} T(\boldsymbol{s}_{j-1}, \boldsymbol{i}_{j-1}, \boldsymbol{s}_j) \\ &\wedge \left(\bigvee\nolimits_{(\varphi, \varphi') \in E_{target}} \widehat{\varphi'}(\boldsymbol{s}_{J_\varphi + k}) \right) \end{aligned} \tag{2}$$

We iteratively check this formula using incremental SAT solving, 'removing' the respective terms from the formula each time a solution satisfies (φ, φ'), until the formula becomes unsatisfiable. In addition to assumptions on the inputs, T must also contain a state invariant, obtained, e.g. with a static analyser. This is necessary because, otherwise, the state satisfying φ in Eq. 2 might be unreachable from an initial state. An imprecise, over-approximating state invariant leads to weights that underestimate the distances between properties. In this case, our algorithms will still compute correct test case chains, but optimization is impaired and more effort may be spent in chain repair.

For the chain repair *RepairPath* (see Algorithm 11), the most efficient method that we tested was to sequentially find a feasible weight for each of the edges in *failed_path*, starting the check for an edge $(\varphi_j, \varphi_{j+1})$ from a concrete state in φ_j obtained from the successful check of the previous edge $(\varphi_{j-1}, \varphi_j)$. However, this method is only complete for strongly connected systems. Hence, in case of failure, we have to try to repair the path by the less efficient method of iteratively calling *CheckPath* on prefixes of *failed_path* with increasing weights (function *RepairPath'*, Algorithm 12, called in line 11 in Algorithm 11).

6 Experimental evaluation

Implementation. For our experiments, we have set up a tool chain (Fig. 8) that generates C code from SIMULINK models using the GENE- AUTO[3] code generator. Our test case chain generator CHAINCOVER[4] itself is built upon the infrastructure provided by CBMC[5] [15] with MINISAT[6] as a SAT backend, the LKH TSP solver[7] [16], and the CLINGO ASP solver[8] [12].

The properties are written in C using the `assert` and `__CPROVER_assume` macros. For instance, property p_1 in our example is stated as follows:

```
void p_1(t_input* i, t_state* s) {
  __CPROVER_assume(s->mode==ON && s->speed==1 && i->dec);
  compute(i,s);
  assert(s->speed==1);
}
```

Assumptions on the inputs and the state invariant obtained from the static analysis are written as C code in a similar way.

Benchmarks. Our experiments are based on SIMULINK models, mainly from automotive industry. For some benchmarks, we had the SIMULINK models or at least the generated C code available; for others we only had screenshots from the SIMULINK models, which we had to re-engineer ourselves. Our benchmarks are a simple *cruise* control model [2], a *window* controller,[9] a car *alarm* system,[10] an elevator model [17], and a model of a *robot arm* that can be controlled with a joystick. We generated test case chains for these examples for specifications of different size and granularity. The

[3] http://geneauto.gforge.enseeiht.fr, version 2.4.9.

[4] http://www.cprover.org/chaincover/, version 0.3.

[5] http://www.cprover.org/cbmc/, version 4.5.

[6] http://minisat.se, version 2.2.0.

[7] http://www.akira.ruc.dk/~keld/research/LKH/, version 2.0.2.

[8] http://potassco.sourceforge.net/, version 3.0.5.

[9] http://www.mathworks.co.uk/products/simulink/examples.html.

[10] http://www.mogentes.eu/public/deliverables/MOGENTES_3-15_1.0r_D3.4b_TestTheories-final_main.pdf.

Fig. 8 Tool chain

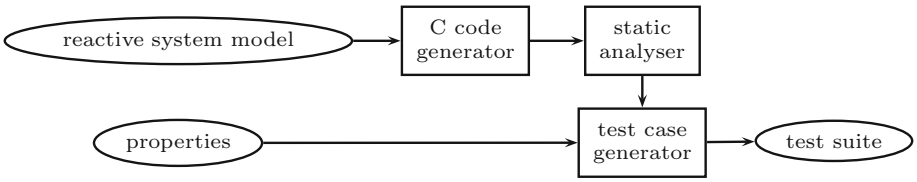


Table 1 Experimental results: the table lists the number of test cases/chains (tcs), the accumulated length of the test case chains (len), and the time (in seconds) taken for test case generation

Benchmark	Size			ChainCover						FShell			Random		
				With Lkh			With Clingo								
	s	i	P	tcs	len	Time	tcs	len	Time	tcs	len	Time	tcs	len	Time
Cruise 1	3b	3b	4	1	9	0.53	1	9	0.88	3	18	3.67	2.8	24.6	0.54
Cruise 2	3b	3b	9	1	10	0.41	4	10	1.74	4	20	3.56	2.4	21.2	0.07
Window 1	3b+1i	5b	8	1	24	7.04	1	24	6.77	4	32	19.0	1.8	40.4	58.9
Window 2	3b+1i	5b	16	1	40	9.89	t.o.			7	56	28.3	2.0	86.8	18.7
Window Ext 1	4b+1i	6b	12	2	30	89.6	2	27	88.1	4	32	38.5	33 % cov.		t/o
Window Ext 2	4b+1i	6b	20	2	50	192	t.o.			6	48	53.1	30 % cov.		t/o
Window Ext 3	4b+1i	6b	4	1	5	8.15	1	5	6.02	2	6	1.73	25 % cov.		t/o
Window Ext 4	4b+1i	6b	8	1	18	72.1	1	20	86.4	4	24	10.5	25 % cov.		t/o
Window Ext 5	4b+1i	6b	9	1	29	269	1	27	239	5	40	27.3	22 % cov.		t/o
Alarm 1	4b+1i	2b	5	1	15	1.95	1	19	5.45	1	27	509	80 % cov.		t/o
Alarm 2	4b+1i	2b	16	1	70	22.8	t.o.			3	81	690	94 % cov.		t/o
Alarm 3	4b+1i	2b	14	2	60	96.4	2	59	189	3	51	106	2	154	1.41
Elevator 1	6b	3b	4	1	8	24.3	1	9	39.1	2	15	115	2.2	10.4	0.85
Elevator 2	6b	3b	10	1	31	155	1	30	237	5	54	789	2.6	49.0	65.8
Elevator 3	6b	3b	19	1	42	491	t.o.			6	54	838	4.0	149	18.0
Robotarm 1	4b+2f	3b	4	1	25	109	1	25	116	2	22	362	2.4	49.0	0.07
Robotarm 2	4b+2f	3b	10	1	47	108	t.o.			2	33	532	3.8	72.2	0.21
Robotarm 3	4b+2f	3b	18	1	85	392	t.o.			5	55	731	3.2	160	0.62

Size indicates the size of the program in the number of (minimally encoded) Boolean (b), integer (i) and floating point (f) variables and (minimally encoded) Boolean (b) inputs. 'P' is the number of properties in the specification. Timed out ('t/o') after 1 h; and the achieved coverage ('cov')

benchmark characteristics are listed in Table 1. Apart from *Cruise 1* all specifications have properties with multi-state antecedents; thus, the obtained test case chains are not minimal, in general. All our benchmarks are (almost) strongly connected (some have an initial transition after which the system is strongly connected). Hence, to evaluate the abstraction refinement and the generation of multiple chains, we added an extension to the *Window* benchmark. Benchmarks *Window Ext* 1–2 and *Alarm* 3 require two chains to cover the properties; benchmarks *Window Ext* 3–5 force our tool to use abstraction refinement (or path constraints).

Comparison. We have compared our tool ChainCover with

1. FShell[11] [18,19], an efficient test generator with test suite minimization, and
2. an in-house, simple *random* case generator with test suite minimization.

We have also compared to Klee[12] [20], a test case generator based on symbolic execution, but the results suggested that its exhaustive exploration is not suitable for our problem.

Like our tool, FShell is based on bounded model checking. FShell takes a coverage specification in form of a query as input. It computes test cases that start in *Init*, cover one or more properties $p_1, \ldots, p_n$ and terminate in *Final* when given the query: `cover (@CALL(p_1) | ... | @CALL(p_n)) -> @CALL(final)`. In the best case, FShell returns a single test case, i.e. a test chain. We have run FShell with increasing unwinding bounds K until all properties were covered.

[11] http://forsyte.at/software/fshell/, version 1.4.

[12] http://klee.llvm.org/.

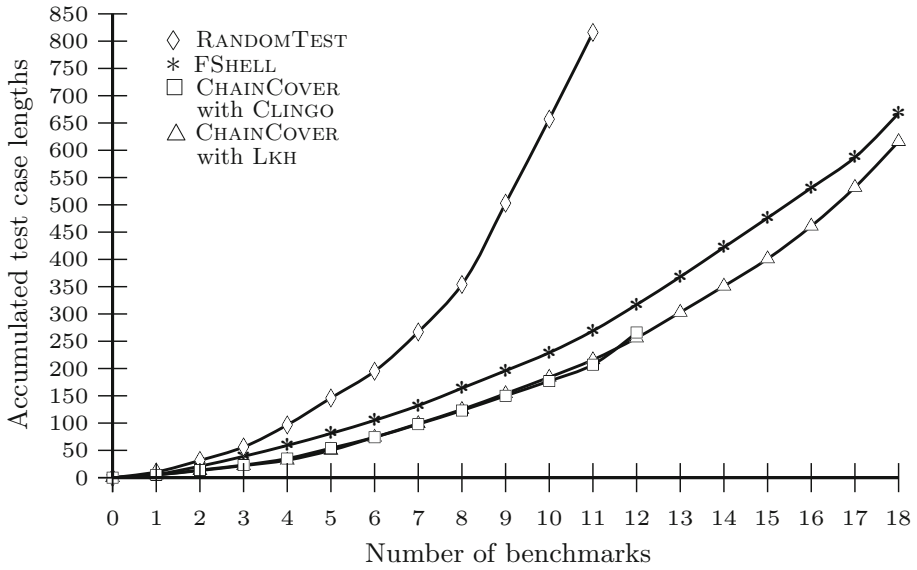


Fig. 9 Experimental results: accumulative graph of test case lengths

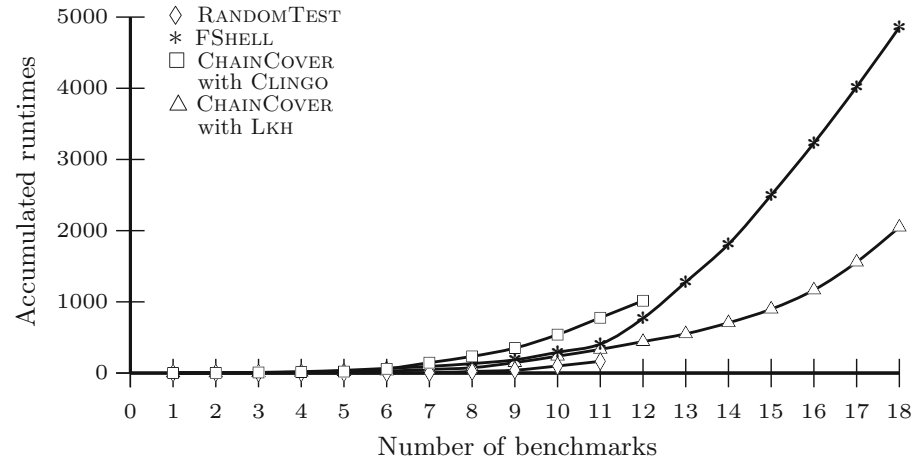


Fig. 10 Experimental results: accumulated runtimes

For random testing, we coded the requirement to finish a test case in *Final* with the help of flags in the test harness. Then we stopped the tool as soon as full coverage was achieved and selected the test cases achieving full coverage while minimizing the length of the input sequence using an in-house, weighted-minimal-cover-based test suite minimiser. We averaged the results over five runs. Unlike ChainCover and FShell, which start test chain computation without prior knowledge of how many transitions are needed to produce a test case, we had to provide random testing with this information. The reason is that the decision when a certain number of transitions will not yield a test case can only be taken after reaching a timeout for random testing. Consequently, the results for random testing are not fully comparable to those of the other tools.

Results. Experimental results obtained are shown in Table 1 and Figs. 9, 10 and 11.

1. Our tool ChainCover usually succeeds in finding fewer and shorter test case chains than the other tools. It is also in general faster. ChainCover spends more than 99 % of its runtime with BMC. The runtime ratio for generating the property K-reachability graph ($\mathcal{O}(Kn^2)$ BMC queries for n properties) versus finding and repairing a chain ($\mathcal{O}(Kn)$ BMC queries) varies between 7:92 and 75:24.
2. With Lkh, the time for solving the ATSP problem is negligible for the number of properties we have in the specifications, whereas Clingo struggles with specifications with more than 10 properties and does not finish within an hour. Abstraction refinement and path constraints seem to perform similarly.
3. FShell comes closest to ChainCover with respect to test case chain length, and finds shorter chains on the robot arm example. However, FShell takes much longer: the computational cost depends on the number of unwind-

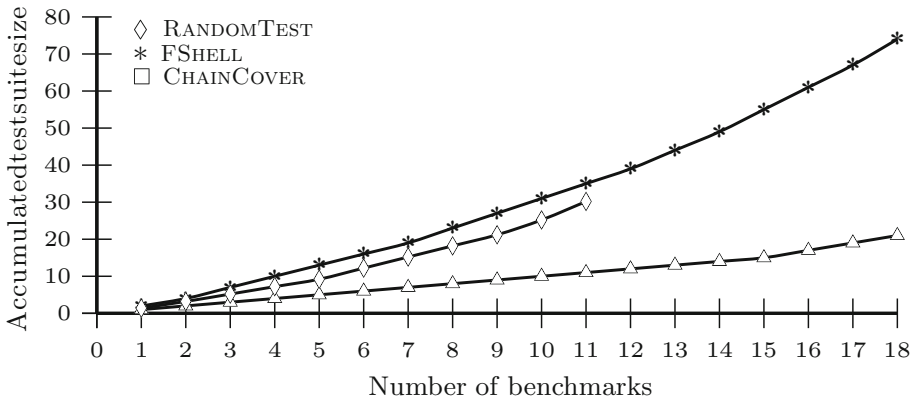


Fig. 11 Experimental results: accumulated test suite sizes

ings and the size of the program and less on the number of properties.

4. Random testing yields very good results on some (small) specifications and sometimes even finds chains that are as short as those generated by ChainCover. However, the results vary and heavily depend on the program and the specification: in some cases, e.g. *Robotarm*, full coverage is achieved in fractions of a second; in other cases, full coverage could not be obtained before reaching the timeout of 1 h and generating millions of test cases.

7 Related work

Test case generation with model checkers came up in the mid-1990s and has attracted continuous research interest since then, especially due to the enormous progress in SAT solver performance. There is a vast literature on this topic, surveyed in [21], for example. The FShell tool [18,19] we have compared with was developed with the motivation of enabling the flexible specification of the desired coverage.

Reactive system testing. There are many approaches to reactive system testing: While random testing [22] is still commonly used, approaches have been developed that combine random testing with *symbolic and concrete execution* (Dart [23], Cute [24], Klee [20]) to guide exhaustive path enumeration. *Scenario-based testing* employ test specifications to guide test case generation towards a particular functionality (e.g., Lutess [25], Lurette [26], Lutin [27]). These methods restrict the input space using static analysis and apply (non-uniform) random test case generation. *Model-based testing* (see [28,29] for surveys on this topic) considers specification models based on labelled transition systems. For instance, extended finite state machines (EFSM) [30–32] are commonly used in communication protocol testing to provide exhaustive test case generation for conformance testing. Available tools include, e.g., Tgv [33] and TorX [34].

Minimal checking sequences and test optimization. In the model-based testing domain, the problem of finding minimal checking sequences has been studied in *conformance testing* [1,3–6], which amounts to checking whether each state and transition in a given EFSM specification is correctly implemented. First, a minimal checking path is computed, which might be infeasible due to the operations on the data variables. Subsequently, random test case generation is applied to discover such a path, which might fail again. Duale and Uyar [35] propose an algorithm for finding a feasible transition path, but it requires guards and assignments in the models to be linear. Another approach is to use genetic algorithms [3,36] to find a feasible execution of minimized length. Also in our setting, the use of genetic algorithms to find minimized instead of minimal solutions is an option to consider. SAT solvers have also been used to compute (non-minimal) checking sequences in FSM models [37,38]. Our method does not impose restrictions on guards and assignments and implicitly handles low-level issues such as overflows and the semantics of floating-point arithmetic in finding feasible test cases. The fact that minimal paths on the abstraction might not be feasible in the concrete program does not arise due to limited reasoning about data variables, but due to the multi-state nature of the properties we are trying to cover.

Closest to our work is a recent work [39] on generating test chains for EFSM models with timers. They use SMT solvers to find a path to the nearest test goal and symbolic execution to constrain the search space. If no test goal is reachable they backtrack to continue the search from an earlier state in the test chain. Their approach represents a greedy heuristics and thus makes minimality considerations difficult. Our method can handle timing information if it is explicitly expressed as counters in the program.

Petrenko et al. [40] propose a method for test optimization for EFSM models with timers. They use an ATSP solver to find an optimal ordering of a given set of test goals and an SMT solver to compute a corresponding test case. Additionally, they take into account overlappings of these test goals. The main differences to our work are that they operate on explicit state machines and their test goals are transition sequences, whereas in our case state machines are implic-

itly given as programs and test goals are sets of transition sequences.

Our approach starts from a partial specification given by a set of properties, usually formalized from high-level requirements. The K-reachability graph abstraction can be viewed as the generation of a model from a partial specification and automated annotation of model transitions with timing information in terms of the minimal number of transitions required.

8 Summary and prospects

We have presented a novel approach to discovering a minimal test case chain, i.e., a single test case that covers a given set of test goals in a minimal number of execution steps. Our approach combines reachability analysis to build an abstraction, TSP-based optimization and heuristics to find a concrete solution in case we cannot guarantee minimality. The test goals might also be generated from an EFSM specification or from code coverage criteria like MC/DC. This flexibility is a distinguishing feature of our approach that makes it equally applicable to model-based and structural coverage-based testing. In our experimental evaluation, we have shown that our tool ChainCover outperforms state-of-the-art test suite generators. Moreover, our approach is not restricted to C code generated from Simulink—it can be applied to any reactive system language. For instance, we could also consider consider Verilog, or the application to HW/SW-co-verification combing Verilog and C code.

Prospects. Deep loops pose a problem for BMC-based methods. For instance, we had to reduce size of loop bound constants in the car *alarm* system benchmark to make it tractable for comparison. Acceleration methods, e.g. [41], are expected to remedy many such situations, especially those involving counters.

Moreover, the property K-reachability graph generation lends itself to parallelization. This is expected to give a further boost to the capacity of our tool.

Test case chains are intended to demonstrate conformance in late stages of the development cycle, especially in acceptance tests when the system can be assumed stable. It is an interesting question in how far they can be used in earlier phases: The test case chains computed by our method are able to cover subsequent test goals even if the implementation has been modified, as long as these modifications have only local effect. Otherwise, it would be desirable to incrementally adapt the test case chain after bug fixes and code changes, for example, by patching parts of the chain. But the practicality and limitations of such an algorithm remain to be demonstrated, and the problem seems very challenging in general. Small changes to the implementation may invalidate large sections of the test case chain, and in the worst case force us to recompute the whole chain. A compositional approach seems to be required, but it remains for future research to investigate the feasibility of this.

Acknowledgments We are grateful to anonymous reviewers for their invaluable comments that helped us substantially improving the paper.

References

1. Hierons, R., Ural, H.: Generating a checking sequence with a minimum number of reset transitions. ASE **17**, 217–250 (2010)
2. Robert Bosch GmbH: Bosch Automotive Handbook. Bentley, Plochingen (2007)
3. Nuñez, A., Merayo, M., Hierons, R., Núñez, M.: Using genetic algorithms to generate test sequences for complex timed systems. Soft Comput. **17**, 301–315 (2013)
4. Petrenko, A., da Silva Simão, A., Yevtushenko, N.: Generating checking sequences for nondeterministic finite state machines. In: ICST, pp. 310–319 (2012)
5. Hierons, R., Ural, H.: Optimizing the length of checking sequences. Trans. Comput. **55**, 618–629 (2006)
6. Hierons, R.: Using a minimal number of resets when testing from a finite state machine. Inf. Proc. Lett. **90**, 287–292 (2004)
7. Schrammel, P., Melham, T., Kroening, D.: Chaining test cases for reactive system testing. In: International Conference on Testing Software and Systems, vol. 8254, pp. 133–148 (2013)
8. Boyd, S., Ural, H.: On the complexity of generating optimal test sequences. Trans. Softw. Eng. **17**, 976–978 (1991)
9. Floyd, R.: Algorithm 97: shortest path. Commun. ACM **5**, 345 (1962)
10. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: VMCAI. LNCS, vol. 2575, pp. 298–309 (2003)
11. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. ENTCS **66**, 160–177 (2002)
12. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: the Potsdam answer set solving collection. AI Commun. **24**, 107–124 (2011)
13. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**, 7–34 (2001)
14. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Formal Methods in Computer-Aided Design, pp. 181–188 (2010)
15. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176 (2004)
16. Helsgaun, K.: An effective implementation of the Lin–Kernighan traveling salesman heuristic. Eur. J. Oper. Res. **126**, 106–130 (2000)
17. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: TAP. LNCS, vol. 6706, pp. 134–151 (2011)
18. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: systematic test case generation for dynamic analysis and measurement. In: CAV. LNCS, vol. 5123, pp. 209–213 (2008)
19. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: VMCAI. LNCS, vol. 5403, pp. 151–166 (2009)
20. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224 (2008)
21. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. **19**, 215–261 (2009)

22. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. Trans. Softw. Eng. **10**, 438–444 (1984)
23. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)
24. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: CAV. LNCS, vol. 4144, pp. 419–423 (2006)
25. du Bousquet, L., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: A specification-driven testing environment for synchronous software. In: ICSE, pp. 267–276 (1999)
26. Jahier, E., Raymond, P., Baufreton, P.: Case studies with Lurette V2. STTT **8**, 517–530 (2006)
27. Raymond, P., Roux, Y., Jahier, E.: Lutin: A language for specifying and executing reactive scenarios. EURASIP J. Embed. Syst. (2008). doi:10.1155/2008/753821
28. Petrenko, A., da Silva Simão, A., Maldonado, J.C.: Model-based testing of software and systems: recent advances and challenges. STTT **14**, 383–386 (2012)
29. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proc. IEEE **84**, 1090–1123 (1996)
30. Lee, D., Yannakakis, M.: Optimization problems from feature testing of communication protocols. In: International Conference on Network Protocols, vol. 66 (1996)
31. Ural, H., Yang, B.: A test sequence selection method for protocol testing. IEEE Trans. Commun. **39**, 514–523 (1991)
32. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. Trans. Softw. Eng. **30**, 29–42 (2004)
33. Jard, C., Jéron, T.: TGV: theory, principles and algorithms: a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. STTT **7**, 297–315 (2005)
34. Tretmans, J.: Model based testing with labelled transition systems. In: Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38 (2008)
35. Duale, A., Uyar, M.Ü.: A method enabling feasible conformance test sequence generation for EFSM models. IEEE Trans. Comput. **53**, 614–627 (2004)
36. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: ICST, pp. 230–239 (2009)
37. Jourdan, G.V., Ural, H., Yenigün, H., Zhu, D.: Using a SAT solver to generate checking sequences. In: International Symposium on Computer and Information Sciences, pp. 549–554 (2009)
38. Mori, T., Otsuka, H., Funabiki, N., Nakata, A., Higashino, T.: A test sequence generation method for communication protocols using the SAT algorithm. Syst. Comput. Jpn **34**, 20–29 (2003)
39. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: NASA Formal Methods. LNCS, vol. 6617, pp. 298–312 (2011)
40. Petrenko, A., Dury, A., Ramesh, S., Mohalik, S.: A method and tool for test optimization for automotive controllers. In: Software Testing, Verification and Validation Workshops, pp. 198–207 (2013)
41. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. In: CAV. LNCS, vol. 8044, pp. 381–396 (2013)